NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

APPROXIMATING THE CHROMATIC NUMBER OF AN ARBITRARY GRAPH USING A SUPERGRAPH HEURISTIC

by

Loren G. Eggen

June 1997

Advisor: Craig W. Rasmussen
Second Reader: Harold M. Fredricksen

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June, 1997	Master's Thesis

4. TITLE AND SUBTITLE APPROXIMATING THE CHROMATIC NUMBER OF AN ARBRITARY GRAPH USING A SUPERGRAPH HEURISTIC

5. FUNDING NUMBERS

6. AUTHORS Eggen, Loren G.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5216

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT(*maximum 200 words*)

We color the vertices of a graph G, so that no two adjacent vertices have the same color. We would like to do this as cheaply as possible. An efficient coloring would be very helpful in optimization models, with applications to bin packing, examination timetable construction, and resource allocations, among others. Graph coloring with the minimum number of colors is in general an NP-complete problem. However, there are several classes of graphs for which coloring is a polynomial-time problem. One such class is the chordal graphs. This thesis deals with an experimental algorithm to approximate the chromatic number of an input graph G. We first find a maximal edge-induced chordal subgraph H of G. We then use a completion procedure to add edges to H, so that the chordality is maintained, until the missing edges from G are restored to create a chordal supergraph S. The supergraph S can then be colored using the greedy approach in polynomial time. The graph G now inherits the coloring of the supergraph S.

14. SUBJECT TERMS Chordal Graphs, Edge Completion Sequences, Elimination Orderings

15. NUMBER OF PAGES 76

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18 298-102

APPROXIMATING THE CHROMATIC NUMBER OF AN ARBITRARY GRAPH USING A SUPERGRAPH HEURISTIC

Loren G. Eggen
Captain, United States Army
B.A., Saint Cloud State University, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 1997

ABSTRACT

We color the vertices of a graph G, so that no two adjacent vertices have the same color. We would like to do this as cheaply as possible. An efficient coloring would be very helpful in optimization models, with applications to bin packing, examination timetable construction, and resource allocations, among others. Graph coloring with the minimum number of colors is in general an NP-complete problem. However, there are several classes of graphs for which coloring is a polynomial-time problem. One such class is the chordal graphs. This thesis deals with an experimental algorithm to approximate the chromatic number of an input graph G. We first find a maximal edge-induced chordal subgraph H of G. We then use a completion procedure to add edges to H, so that the chordality is maintained, until the missing edges from G are restored to create a chordal supergraph S. The supergraph S can then be colored using the greedy approach in polynomial time. The graph G now inherits the coloring of the supergraph S.

DISCLAIMER

The computer programs in the Appendices are supplied on an "as is" basis, with no warrantees of any kind. The author bears no responsibility for any consequences of using these programs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ACRONYMS

$\|V\|$	The *cardinality* of a set V.
G=(V, E)	The *graph* G with vertex set V and the edge set E.
uv	An *edge* between the vertices u and v.
$Adj(v)$	The *adjacency* set of vertex v.
$N(v)$	The *neighborhood* of vertex v;$N(v) = \{v\} + Adj(v)$.
$\overline{G}$	The *complement* of an undirected graph G.
G_S	The *subgraph* of G *induced* by S.
K_n	The *complete graph* on n vertices.
$\alpha(G)$	The *independence number* of G.
$\beta(G)$	The *vertex cover number* of G.
$\Delta(G)$	The maximum *vertex degree*.
$\chi G)$	The *chromatic number* of G.
$\omega(G)$	The *clique number* of G.
$O(f(m))$	Computational complexity *on the order of* $f(m)$.
P	The class of *deterministic* polynomial-time problems.
NP	The class of *nondeterministic* polynomial-time problems.

ASP	Ammunition Supply Point
BFS	Breadth-first search
DFS	Depth-first search
MCS	Maximum cardinality search
PEO	Perfect elimination ordering

ACKNOWLEDGMENTS

I would like to express my appreciation of the support and guidance received from the faculty, staff, and students in the Mathematics Department at the Naval Postgraduate School. I am most grateful to Craig W. Rasmussen for his helpfulness and generosity.

I. INTRODUCTION

Large-scale scheduling and timetable problems arise in many activities, from crew scheduling on airlines to scheduling of classroom periods and teachers at Universities to register allocations in a computer CPU. These are often modeled as graph coloring problems that are then subjected to a variety of strategies for solution by computer. For example, consider the storage problem at military ammunition supply points, where certain types of ammunition cannot be stored in the same bunkers. What is the minimum number of bunkers needed at an ammunition supply point to ensure the safe storage of all types of ammunition? We use a graph-theoretic model to formulate this problem. We construct a graph G, with vertices representing ammunition types and where the existence of edges between vertices represents their pairwise incompatibilities. This problem reduces to that of finding an optimal coloring of the vertices of G. These types of optimization models have applications in bin packing, examination timetable construction, and resource allocations. Unfortunately there is no known algorithm for this problem which will predictably provide an optimal solution in a reasonable time.

A. DEFINITIONS

The graph terminology used in this thesis can be found in most textbooks on graph theory. For undefined terms and notation see Bondy and Murty [Ref. 1], or West [Ref. 2].

1. Undirected Graph

A simple *undirected graph* $G = (V, E)$ consists of a vertex set $V(G) = \{v_1, ..., v_n\}$ and an edge set $E(G) = \{e_1, ..., e_m\}$, where each edge is an unordered pair of vertices (see part (a), Figure 1). We use uv to denote the edge $\{u, v\}$. When we have $uv \in E(G)$, then we say that "u is adjacent to v" and "v is adjacent to u". The vertices of an edge e are its *endpoints*. If the endpoints are the same this edge is

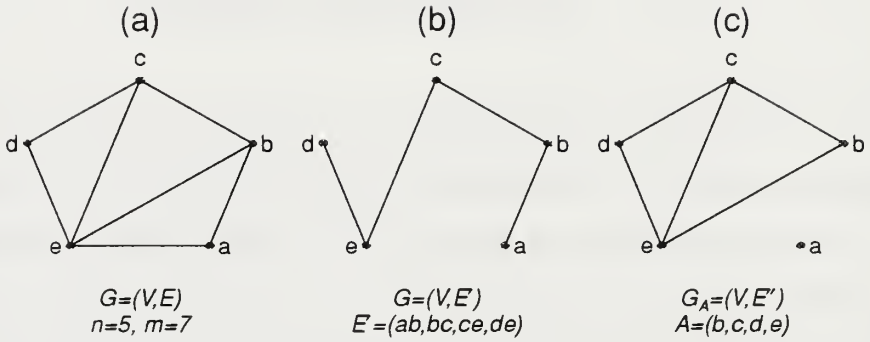

Figure 1. (a) Undirected graph G of order n and size m, (b) A subgraph H with edge set $E\prime$, (c) An induced subgraph with vertex set A.

considered a *loop*. In a graph G, a list of vertices $[v_0, v_1, v_2, \ldots, v_l]$ is a *path* of length l from vertex v_0 to v_l if $\{v_{i-1}, v_i\} \in E(G)$, for all $i = 1, 2, \ldots, l$. A path is called *closed* if $v_0 = v_l$. A *cycle* is a simple closed path of distinct vertices. A graph G is *connected* if between any two vertices there exists a path in G joining them. The *neighborhood* of v is the set of adjacent vertices $Adj\,(v)$ and is denoted $N\,(v)$. We consider only connected, loopless, finite graphs without multiple (redundant) edges.

2. Miscellaneous Graphs

The *order* of a graph G, denoted $|V|$, is the number of vertices in G. We use $|E|$ to denote the number of edges, or the *size* of G. The *complete graph* of order n is denoted K_n (see part (c), Figure 2), and is an undirected graph in which every pair of vertices is adjacent. The *complement* of a graph G is the graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(x, y) \in V | x \neq y \text{ and } (x, y) \notin E\}$ (see part (b), Figure 2). A *subgraph* of G is a graph H, such that $E(H) \subseteq E(G)$, $V\,(H) \subseteq V\,(G)$(see part (b), Figure 1). If $A \subseteq V$, the *A-induced subgraph* of G is the graph G_A, such that $E\,(G_A) = \{xy \in E\,(G)\,| x \in V\,(G_A) \text{ and } y \in V\,(G_A)\}$ (see part (c), Figure 1).

3. Graph Parameters

For the following definitions let $G = (V, E)$ be an undirected graph. A *coloring* of a graph is an assignment of "colors" to the set of vertices V so that adjacent vertices have different colors. The symbol $\chi(G)$ denotes the smallest number of colors

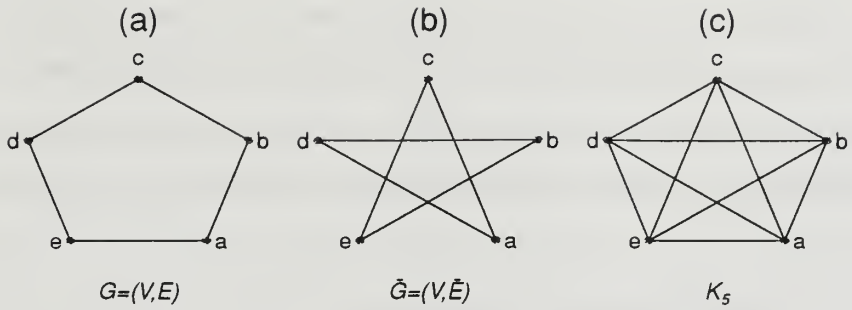

Figure 2. (a) Undirected graph G, (b) Complement graph $\overline{G}$ with edge set $\overline{E}$, (c) Complete graph K_5.

required and is called the *chromatic number* of the graph G(see figure4). A subset $V(H) \subseteq V(G)$, with $|H| = q$, is a *q-clique* if it induces a complete subgraph K_q. A clique H is *maximal* if there is no clique of G which properly contains H as a subset and H is *maximum* if no other clique in G has a larger order. The number of vertices in a maximum clique of G is called the *clique number* of G, and is denoted $\omega(G)$. An *independent set* $S \subseteq V(G)$ is a set of vertices in which no two vertices are adjacent. The number of vertices in a independent set of maximum order is called the *independence number* of G, denoted $\alpha(G)$. A subset $K \subseteq V(G)$ such that if $xy \in E(G)$, then $x \in K$ or $y \in K$, is called a *vertex cover*. A vertex cover for G is a set of vertices that is collectively incident to all the edges in $E(G)$. The *vertex cover number* of G is the number of vertices in the minimum vertex cover, denoted $\beta(G)$.

4. Chordal Graphs

A graph G is *chordal* (or *triangulated*) if it does not contain any cycle of length greater than three as an induced subgraph[Ref. 3]. An *ordering* of the vertex set V, with $n = |V|$, is $\{v_0, v_1, v_2, \ldots, v_n\}$. A vertex v is *simplicial* in G if $N(v)$ is a complete subgraph. If δ is an ordering of V such that each v_i is a simplicial vertex of the induced subgraph $G_{v_i,\ldots,v_n}$, then it is called a *perfect elimination ordering* (see Figure 4). A *successor* of v_i with respect to the ordering δ is a vertex $v_j \in N(v_i)$, where $i < j$, and is denoted $Suc(v_i)$[Ref. 4]. A chordal graph G is of the class of *perfect graphs*,

in which $\omega(G_A) = \chi(G_A)$ for all $A \subseteq V(G)$. The class of perfect graphs have been extensively researched since the 1960's by well-known mathematicians. Golumbic's book "Algorithmic Graph Theory and Perfect Graphs" [Ref. 3] does an excellent job in explaining the difficult practical problems related to the structure of perfect graphs.

5. NP-Complete Problems

The class of problems with complexity bounded by a polynomial in the size of the input is denoted P. We consider a problem solvable in polynomial time by a deterministic algorithm as being tractable. That is, for each input of size n the worst-case running time is $O(n^k)$ for some constant k. We define problems that require superpolynomial time as being intractable.

The class NP contains those decision problems that are "solvable" by a *non-deterministic* polynomial-time algorithm. Such an algorithm, in a sense, tries all possibilities simultaneously, applying polynomial-time computation to each guess in parallel. This type of algorithm should not be confused with a parallel implementation of a deterministic algorithm. If any of the computations results in a yes or possibly a no, then the algorithm is a success. The algorithm is successful if it works, even if the answer to the current decision problem is negative. The non-determinism concerns the multiplicity of paths, and not whether the search is successful. It is easy to see that if we can do many computation paths in parallel and one of these is completed in polynomial time, then we can do that one alone in polynomial time. Therefore $P \subseteq NP$.

Most mathematicians believe that the classes P and NP are different classes, although it has not yet been proven that $P \neq NP$. The class P, loosely, consists of those problems that can be solved quickly, while the class NP consists of problems for which a solution can be *verified* quickly. We define a problem X as *NP-hard* if every problem instance in NP can be reduced to an instance of X in polynomial time. A problem is *NP-complete* if it is in NP and is NP hard. We now have the

class of P in the class of NP, but the problems that are NP-complete are also in NP, so a problem in P is probably not NP-complete. Since no one has come up with a polynomial algorithm for a NP-complete problem, thus proving that $P = NP$, we can assume the intractability of NP-complete problems. Problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete. Thus, it is important to become familiar with this class of problems. For a detailed discussion of NP-completeness see Cormen, Leiserson, and Rivest[Ref. 5] or Garey and Johnson[Ref. 6].

II. EXISTING ALGORITHMS

Ore's book "The Four-color Problem" [Ref. 7] shows that considerable literature in the field of graph theory deals with the coloring of graphs. Many algorithms exist for graph coloring, but there are only a few fundamentally different approaches to the problem. Finding the exact chromatic number of a graph is a NP-complete problem, so it is no surprise that no fast algorithm exists. We will explain two such algorithms and it should then be obvious to the reader why they are impractical.

A. EXACT ALGORITHMS

1. Brute-Force Coloring

The different ways to color a graph are not unique. There may be several proper colorings of graph G using $\chi(G)$ colors. There also exist many improper colorings of that same graph. How many different colorings of G are there? A coloring of the n vertices in V using a palette P of order k is a mapping $f : V \to P$. If we use all of P, the mapping is onto. Then there are

$$\binom{k}{k}k^n - \binom{k}{k-1}(k-1)^n + \binom{k}{k-2}(k-2)^n + \cdots + (-1)^{k-1}\binom{k}{1}1^n$$

different colorings, and if $k \geq \chi(G)$ one or more of these colorings is proper. We could easily program a computer to produce these colorings and for each, check to see if the coloring is proper. But the cost of computation would be exorbitant. Using the formula above we see that the number of colorings gets outrageous very quickly. For example, let $n = 10$, then if $k = 2$ there are 1022 different colorings. If $k = 3$ there are $55,977$ colorings and if $k = 4$ there are $818,521$ different colorings. If we wanted to color a large graph in this way, say $n = 100$, and $k = 2$ there are $1.267 * 10^{30}$different colorings.

2. Greedy-Backtracking Coloring

Given G, this algorithm finds $\chi(G)$ if given a starting palette of fewer than $\chi(G)$ colors. It performs an extensive routine of trying to color the graph with the palette given, and if it determines that the palette is not large enough then another color is added to the palette. Eventually enough colors appear on the palette to successfully color the graph.

The most significant problem with this algorithm is to determine the starting number of colors k. If k is too low, the algorithm consumes too much time backtracking and recoloring the graph and if k is greater than χ then algorithm will greedily color the graph with the available colors k and the resulting coloring may not be optimal. The clique number $\omega(G)$ is clearly a lower bound on $\chi(G)$, since the vertices of the largest complete subgraph of G must all have separate colors. In small graphs, $\omega(G)$ is very close to $\chi(G)$, but in larger graphs the difference can grow significantly, as is shown in the next section.

B. APPROXIMATION ALGORITHMS

Several algorithms exist to approximate the chromatic number of an arbitrary graph. Some are better than others. Most of these algorithms deal with finding the largest independent set of vertices or the vertices of a maximal clique. To find the largest independent set is a NP-complete problem, but to find a large one can be done in polynomial time.

1. Independent Sets

The approximation algorithm for the independent set problem is based on two assumptions: (1) a vertex of high degree is harder to color than a vertex of low degree; (2) coloring many vertices with the same color is good. Recall that $\alpha(G) = \omega(\overline{G})$ is the order of a largest independent set of the vertices in G. Clearly the vertices in an independent set can be colored with the same color, therefore $\chi(G) \leq n/\alpha(G)$, where $n = |V|$. In small graphs, $n/\alpha(G)$ tends to be smaller than $\omega(G)$, but in larger

graphs $n/\alpha(G)$ is a much closer lower bound to $\chi(G)$.

Let $f(n,p) = E(\alpha)$ be a function to find the expected value $\alpha(G)$ for a random graph G of order n and edge probability p. With this information we can estimate χ. In principle we can find an independent set of order $E(\alpha)$, and delete it. Now we have a graph of order $n - f(n,p) = n_1$. We continue on the same way until $E(\alpha) = 0$ and thus $\tilde{\chi}(n,p) = i + \tilde{\chi}(n_i - f(n_i,p),p) \geq n/\alpha(G)$, for $i = 0,1,2,\ldots$. Note: $\tilde{\omega}(G) = \tilde{\alpha}(G)$ for graphs with edge probability $p = .5$ and in particular $\tilde{\chi}(1000,.5) = 85$, but $\tilde{\alpha}(1000,.5) = 15$ and $n/\tilde{\alpha}(1000,.5) = 67$. This points out a very peculiar problem with graphs of large order and density. That is to say, $\omega(G)$ is a very poor estimator while $n/\alpha(G)$ is a good estimator of $\chi(G)$ for graphs of large order and density.

2. Vertex Covers

In a graph G, if a set $A \subseteq V(G)$ is a vertex cover then there are no edges in $\overline{A}$, an independent set. Thus any minimum vertex cover is the complement of a maximum independent set, and so $\alpha(G) + \beta(G) = n$, where n is the order of G. The vertices in a minimum vertex cover are the only vertices considered in the coloring problem; all other vertices are part of the independent set and require only one additional color. Therefore $\beta(G) + 1 \geq \chi(G)$. Another upper bound on $\chi(G)$ that is worth considering is $\Delta(G) + 1$. Both errors can be relatively large, though.

The vertex cover problem is known to be NP-complete (see Cormen[Ref. 5]). Nevertheless, there exist good algorithms to find a vertex cover that is near optimal. One such algorithm uses a set C, initially empty, and a set E containing the edges of a graph G. We pick an arbitrary edge $xy \in E$ and add the vertices x and y to C, then delete any edges in E covered by x or y. We pick another edge in E and continue this procedure until E is empty. The computational complexity of this algorithm is $O(m)$, where m is the size of G. The vertex cover produced by this algorithm is C, which is at most twice the size of the optimal cover C^*. Let A be the set of arbitrary edges picked in the algorithm. No two edges in A share an endpoint, since all incident edges to the endpoints are deleted before the next edge is picked. Therefore when

two vertices are added to C, $|C| = 2|A|$. Any vertex cover of A must contain at least one endpoint of each edge in A. Since no two edges in A share an endpoint, no vertex in the cover is incident on more than one edge in A. Therefore, $|A| \leq |C^*|$, and $|C| = 2|A| \leq 2|C^*|$.

3. Maximum Clique

This algorithm, by Balas and Yu [Ref. 8], is a chordal subgraph approach for finding the maximum clique problem. It has two main subroutines. The first algorithm generates a maximal triangulated induced subgraph H of an arbitrary graph G in a computational complexity of $O(n+m)$. The second finds the minimum coloring of H, using the cardinality k of the maximum clique; then appends vertices to H while maintaining its chromatic number, until the resulting graph becomes a maximal k-chromatic induced subgraph F of G. If $F = G$, we are done, since the maximum clique in H was also maximum in G. Otherwise we branch to subproblems considering any clique larger than the current one must contain one of the vertices in $V(G)$ but not in $V(F)$. We now apply the same procedure above on the new subproblems, each defined on a vertex set contained in the neighbor set of $v \in V(G)/V(F)$. For the results of this algorithm and the different variations applied to it see Balas and Yu[Ref. 8].

4. Minimal Weighted Coloring of Chordal Subgraphs

The algorithm in the previous section was modified by Balas and Xue [Ref. 9] to find the minimum weighted clique and thus a coloring of a chordal subgraph H of G. They then extended the algorithm to include an ordering τ of the vertices $V \backslash H$. This ordering τ is used to add remaining vertices to the correct color class until a maximal induced subgraph F with the same minimum weighted coloring of H results. The final step is to modify the branching rules described by Balas and Yu [Ref. 8] to include the minimum weighted coloring and define the subproblems to reapply in the algorithm until it finds the maximum weighted clique.

5. Edge-Maximal Chordal Subgraph

The algorithm by Xue [Ref. 4] involves n iterations. Each iteration augments the partial perfect elimination ordering and adds a vertex, together with some edges, to the partial chordal subgraph. What sets this algorithm apart from algorithms like that of Dearing et al.[Ref. 10] are the way in which it chooses the next vertex to add into the partial chordal subgraph, and the way in which it chooses the first successor of a given vertex. We use a greedy approach in both instances. During an iteration, we call the vertices in the partial perfect elimination ordering *labeled* and the rest *unlabeled.* Let U be the set of unlabeled vertices and $H = (V(H), E(H))$ be the partial chordal subgraph. For every unlabeled vertex $v \in U$, we assign v a temporary first successor $t(v)$ and a label $s(v)$, where $t(v) \in V(H)$. The label $s(v) = 1 + |N(v) \cap Suc_H(t(v))|$ is the maximum number of edges that can be added into H if v is added into H next with $t(v)$ being its first successor. We initially set $t(v) = \oslash$, no temporary first successor and $s(v) = 0$, $\forall v \in V$. We choose the next vertex $v \in U$ to label and add to H such that $s(v) = \max\{s(u) | u \in U\}$. Ties go to the vertex v with the largest degree. We add all the edges to $t(v)$ or to a vertex in $Suc_H(t(v))$. We update $t(u)$, $s(u)$: For all $u \in N(u) \cap U$, let r_u be the number of neighbors of u in H that are either v or a successor of v. If $r_u < s(u)$, set v as the temporary first successor of u and update $s(u)$, i.e., let $r_u = 1 + |Suc_H(v) \cap N(u)|$. If $r_u \geq s(u)$, set $t(u) = v$, $s(u) = r_u$.

III. A SUPERGRAPH HEURISTIC

The basic idea is, given G, to find S, a minimum cardinality chordal supergraph of G. We could then color S, and let G inherit the result. What makes this a topic worthy of research is the fact that the *first step* is NP-complete. We get around this by finding a minimal chordal supergraph of G, ideally containing as few edges of $\overline{G}$ as possible.

In this chapter we introduce and explain an experimental scheme for coloring an arbitrary graph. We have developed a basic algorithm which will be modified in an attempt to improve its performance. Each experiment consists of the generation of 100 random graphs of order 7 to 20, to which the algorithm is applied. An implementation in the MATLAB programing language is given in the Appendices for all functions whose names appear here in the `verbatim` typeface.

A. THE IDEAL ALGORITHM

The ideal supergraph algorithm for the coloring an arbitrarygraph G consists of finding the closest edge-induced chordal supergraph S of G. We color S with a simple greedy coloring algorithm, capitalizing on the perfect ordering of the chordal graph S, and then let G inherit the coloring. At a minimum we have $\chi(S) \geq \chi(G)$ and if only a few inconsequential edges were induced to find S then possibly $\chi(S) = \chi(G)$. The following algorithm depicts the ideal coloring algorithm.

Ideal Algorithm

Input: Graph G
Ouput: Optimal coloring of a minimal chordal supergraph S
begin
$S :=$ minimum edge-induced supergraph of G;
$\gamma :=$ coloring of S;
H inherits the coloring γ;
end;

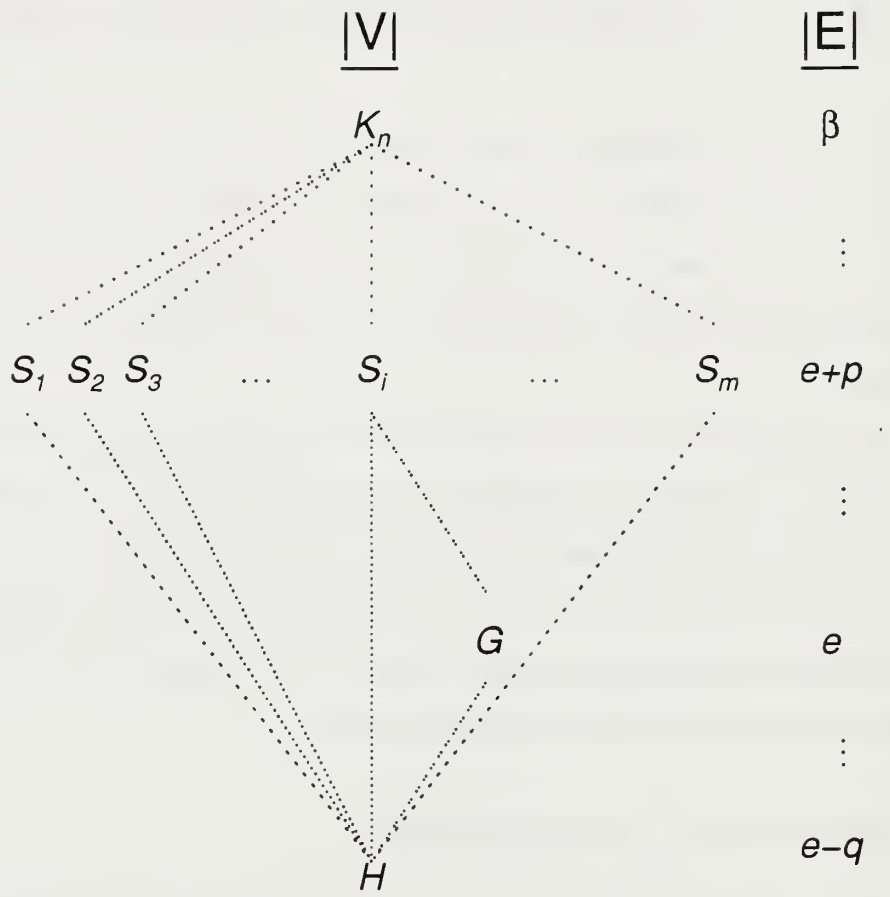

Figure 3. The Ideal Algorithm starts with an arbitrary graph G, computes a maximal chordal subgraph H, and then builds a supergraph S while minimizing p. $(\beta = \binom{n}{2})$

The problem with this algorithm is that of finding the chordal supergraph S. We propose, as depicted in Figure 3, to find a maximal chordal subgraph H of G and then find a chordal supergraph S minimizing the number of edges p.

B. THE BASIC ALGORITHM

This algorithm, implemented in the function `project1.m`, is the basis for the improved algorithms which follow. Generally speaking, it produces a random graph, ensures the graph is connected, and computes its chromatic number(see figure 4). It then finds a maximal chordal subgraph of the random graph and a minimal chordal supergraph of both graphs. Lastly it computes the relative difference between the

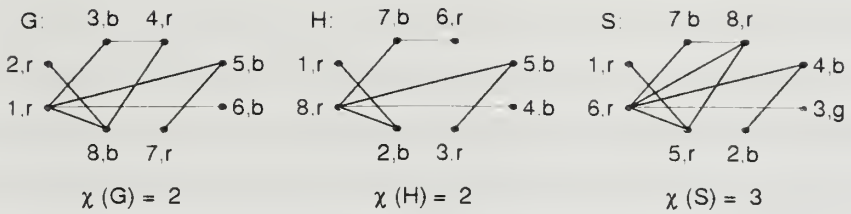

Figure 4. The Basic Algorithm starts with an arbitrary graph G, computes a maximal chordal subgraph H, and then builds a supergraph S. Graphs H and S vertices are in the perfect elimination ordering δ.

chromatic numbers of the supergraph and the original graph.

1. Random Graph Generation

The basic Algorithm A, step 1, generates a random input graph $G = (V, E)$ of order n and size m, with vertices labeled as $v_1, ..., v_n$. A random sparse adjacency matrix is generated by the function `unigraph.m` and is used to represent the undirected graph G. An edge in G exists between the vertices v_i and v_j, represented by a 1 in the (i, j) entry of the adjacency matrix, with probability p, where $0 < p < 1$.

2. Test for Connectedness

The function to ensure that a graph is connected uses a depth-first search algorithm and has computational complexity of $O(n + m)$ as discussed in Roberts [Ref. 11, page 445]. For a detailed discussion of depth-first search see Tarjan [Ref. 12]. Since a graph of order n has at most $\binom{n}{2}$ edges, we have $O(n^2)$ steps. In this function each time we traverse down a path to the end and return to the beginning without visiting every vertex in the graph we identify a connected component of the graph. When a disconnected graph is discovered we discard the graph and return to step one.

The function `connect.m` starts with a list v containing the initial vertex v_1 and visits each vertex using depth-first search. At each vertex v_i there are three possibilities: (1) there are adjacent vertices which have not been added to the list of vertices v, in which case we pick the lowest-indexed vertex, add it to the list v, delete

the edge and continue the search from the new vertex; (2) there are adjacent vertices, but all of them appear in the list v, thus we delete all the edges and backtrack on the list v until we find an adjacent vertex and continue the search; or (3) there are no adjacent vertices, in which case we backtrack on the list v as described in case two. If in the process of backtracking to find an adjacent vertex we end up at the initial vertex v_1 and $|v| \neq n$, then we have found a connected component in a disconnected graph.

3. Maximal Chordal Subgraph Computation

A maximal chordal subgraph H of the input graph G is computed in step two of the basic algorithm. H is found by using an algorithm of Dearing, Shier and Warner [Ref. 10]. This is a polynomial algorithm used in optimization problems to solve large systems of linear equations. The algorithm has worst-case time complexity $O(m\Delta)$, where Δ denotes the maximum vertex degree in G. The `mchord.m` function is given a starting vertex v_1 and the sparse adjacency matrix for G, and produces a list of vertices denoting a perfect elimination ordering and a list of edges for the maximal chordal subgraph found.

The function `mchord.m` generates a perfect elimination ordering list v, starting with the first vertex given. It maintains a list s of all the vertices of G not in v and a list E of all the edges in the maximal chordal subgraph. It also builds an incidence matrix N of order nxn containing a 1 in the (i, j) entry if vertex i is adjacent to vertex j when both vertices have been considered in the maximal chordal subgraph. The matrix N is initialized to all zeros and list E is empty at the beginning of the procedure. The next step in the `mchord.m` function is to generate a loop to visit each vertex in graph G. The current vertex is denoted v_0. Step one in the loop is to find all the vertices adjacent to v_0, using a function `adj.m`.

The function `adj.m` is given the adjacency matrix G and a vertex v_0. The function determines the vertices adjacent to v_0 by examining the ith row, representing v_0, of the matrix G and returning the indices of all columns j which include a 1 in

the (i, j) position. The function returns a list u of column indices representing the adjacent vertices and a list E of edges incident to v_0.

For each vertex u that is adjacent to the current vertex v_0, if $N(u) \subseteq N(v_0)$ then $N(u) := N(u) \cup \{v_0\}$ and $E := E \cup \{u, v_0\}$. In other words, if the neighborhood of u is a subset of the neighborhood of v_0 we increment the value for the vertex u in the set N and add the edge to E. At this time we eliminate the edges which have been added to the subgraph from the adjacency matrix G, so that they will not be considered again later.

From the set N we choose the vertex with the largest value to become our new v_0 and we add it to the list of v and eliminate it from the list of s. Now we repeat the procedure until all the vertices in G are added to the list v. The result is the reversal of a perfect elimination ordering of G.

4. Missing Edges

The functions `mkadjmat.m` and `mkedges.m` are used to derive an adjacency matrix from a list of edges and to create a list of edges from an adjacency matrix, respectively. When we subtract the adjacency matrix H from the adjacency matrix G, we produce an adjacency matrix F representing the edges of G missing from H. The list of missing edges is denoted $em = (\{v_i, v_j\} | v_i \text{ is adjacent to } v_j \text{ and } v_i, v_j \in F)$. Anytime we reorder the vertices of H we must translate the list of missing edges em into the new ordering using the function `trans.m`.

5. Maximum Clique

The clique number $\omega(H) = \chi(H)$, since H is a chordal Perfect graph. It is trivial that $\chi(H) \leq \chi(G)$. Therefore, we use $\omega(H)$ as a lower bound on $\chi(G)$, because it is the easiest to find. The algorithm to find the maximum clique of a chordal graph by Gavril [Ref. 13], has a computational complexity of $O(n + m)$. We use this algorithm in the function `mclique.m` to determine the $\omega(H)$. We maintain a list S which holds the number of times the lowest-indexed vertex was a member

of a previous clique. As we visit each vertex in the perfect elimination ordering we determine its neighbors and store them in x. If $x = \emptyset$ then the current vertex has no neighbors and it is its own maximal clique. If $x \neq \emptyset$ then we only need to consider the neighbors of higher index in the perfect elimination ordering since all vertices of lower order would have been eliminated and not considered in finding the next maximal clique. We store these vertices in X. We now update the value in S for the smallest-indexed vertex u in X by $S(u) = \max\{S(u), |X| - 1\}$. If the number in S of the current vertex is less than the current maximal clique, we output the maximal clique X and update the maximal clique number if the current maximal clique is larger. If the number in S is equal to or greater than current clique number, then the maximal clique has already been identified and we continue. In layman's terms, we visit each vertex in the perfect elimination ordering, cutting off the portion of the graph we just visited. We then look forward, relative to the ordering, and determine the maximal clique. If it is larger than the current clique number we update the clique number and move on until we reach the end of the perfect elimination ordering.

6. Greedy-Backtracking Coloring Scheme

The greedy-backtracking coloring scheme uses an algorithm defined by Bender and Wilf in [Ref. 14]. They give a detailed analysis of the run-time complexity of this algorithm on arbitrary graphs. The idea is to visit each vertex in the order given and determine which of its neighbors have been colored. We always start with the cheapest (or lowest) color and, having greedily colored the first k vertices, find the cheapest available to color the current $(k + 1)$st vertex. This algorithm must be given a palette of available colors, and if in visiting a current vertex we run out of available colors, we simply backtrack to the last vertex colored and determine if it is possible to increase the color to a (the next highest). If it is possible to increase the color we do and continue our search. If it is not possible we must backtrack further, erasing the current coloring scheme until we find a vertex color which can be increased. If in the backtracking process we return to the initial vertex, then we did not supply a

sufficient number of colors in the palette. In this case we must increase the number of colors available on the palette and try the procedure again. The algorithm terminates with a coloring that uses the least number of colors necessary for an proper coloring scheme of the graph. This least number of colors is of course $\chi(G)$.

7. Maximum Cardinality Search

The vertices of chordal subgraph H are ordered in the original perfect elimination ordering returned by the `mchord.m` function and, consequently, most of the missing edges are at the front of the ordering. It would be more beneficial for our algorithm to have the missing edges towards the back of the ordering, since these are the vertices considered early in the edge completion scheme. We use the maximum cardinality search (MCS) algorithm described in Tarjan[Ref. 15], to reorder the vertices of the subgraph H into a new perfect elimination ordering. The computational complexity of this algorithm is $O(n+m)$. The function `mcs.m` uses a list x of size $|V|$ representing each of the vertices, which is initially set to zeros. Each time a vertex visited all of its neighbors the value on list x is increased by one. We use the vertex with the highest number on the list x as our next vertex to visit. We add this vertex to the list v and then continue the search. The algorithm terminates when all the vertices have been added to v. The list v represents the perfect elimination ordering of H. This new ordering ends with the first vertex of the old perfect elimination ordering, and is still a perfect elimination ordering since the subgraph H is chordal.

8. Edge Completion

We now use a procedure first described by Grone, Johnson, et al in [Ref. 16] to perform a edge completion sequence on the chordal graph until all missing edges have been reinserted. This results in a supergraph of the input graph. The computational complexity of this procedure is $O(n^2)$. In the function `complete.m` we start at the last vertex in the perfect elimination ordering and connect it to the remaining vertices starting with the next highest. Each time an edge in added to the graph we check to

see if it is on the list of missing edges. If it is on the list of missing edges we eliminate it from the list and continue the procedure until the list is empty.

9. Greedy Coloring

Finally we perform a procedure on the chordal supergraph to determine its chromatic number with the function `grcolor.m`. This function takes advantage of the input chordal graph in a perfect elimination ordering. Since S is a perfectly orderable graph, applying the greedy coloring algorithm produces an optimal coloring of the graph in $O(n)$. Each vertex is colored using the cheapest available color until all have been colored.

C. IMPROVED ALGORITHMS

With these improved algorithms we want to reduce the relative difference in the chromatic numbers of the original arbitrary graph and the chordal supergraph. First we experiment with manipulations on the arbitrary graph and then we examine some special ordering of the chordal graphs.

1. Vertex Sort Algorithm

This variation is based on the assumption that a better perfect elimination ordering is produced for the subgraph H when the vertices of G are pre-sorted by degree, highest first. This perfect elimination ordering might facilitate the selection of the supergraph S that would minimize the amount of added edges. In this variation we sort the vertices of the arbitrary graph G with the function `versort.m`. The vertices with the highest degree are considered first in the function `mchord.m`. Thus, the resulting maximal chordal subgraph H has an improved perfect elimination ordering. This algorithm is incorporated in the function `project2.m`.

2. Missing Edge Algorithm

This variation presupposes that a better perfect elimination ordering is used on the maximal chordal subgraph H if the vertices of H are presorted so that the vertices

with the most missing edges are considered first in the edge completion algorithm. The resulting supergraph S should contain the minimal number of unnecessary edges.

D. COMPUTATIONAL COMPLEXITY

The most time-consuming function in this algorithm is the greedy-backtracking coloring function `gbcolor.m`, which is an inefficient non-polynomial time algorithm. All other functions in these algorithms run in polynomial time (see Table I). The inefficiency of `gbcolor.m` has limited us to graphs of order 20 or less. We presume another limiting factor in this algorithm is the programming language MATLAB.

MATLAB Function	Worstcase Run-time
`unigraph1.m`	$O(n^2)$
`connect1.m`	$O(n+m)$
`mchord.m`	$O(m\Delta)$
`mclique.m`	$O(n+m)$
`gbcolor.m`	non-polynomial time
`mcs.m`	$O(n+m)$
`complete.m`	$O(n^2)$
`grcolor.m`	$O(n)$

Table I. Computational Complexity.

IV. EXPERIMENTAL RESULTS

The algorithms in chapter III were implemented in MATLAB and tested on 100 random graphs of various densities (where density is the probability of an edge existing between any two vertices), having an order of 7 to 20. The projects were run on a HP700/15 workstation. Table II summarizes the results. The results are stated in the framework of the relative error between computing the actually chromatic number of the graph G and the chromatic number of the chordal supergraph S.

A. GRAPH ORDER

As can be seen from Table II and Figures 5 - 7, relative error increases with the order of the graph. Problem difficulty for our algorithms increases as well, which is not peculiar to our approach, but is intrinsic to the nature of the problem. Graphs

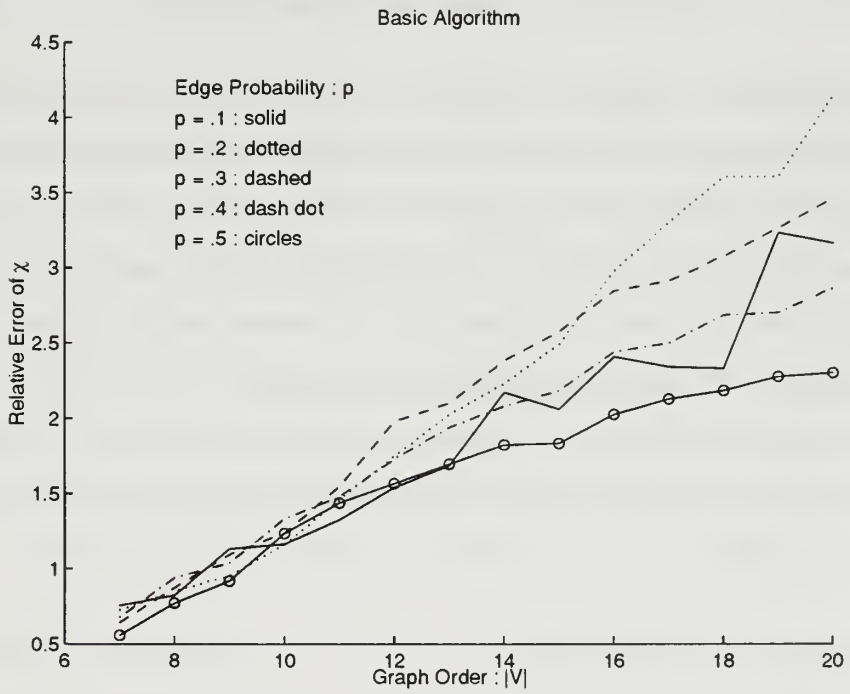

Figure 5. Relative error of χ versus graph order. Note: This is discrete data. Relative error has been represented by continuous lines for clarity only.

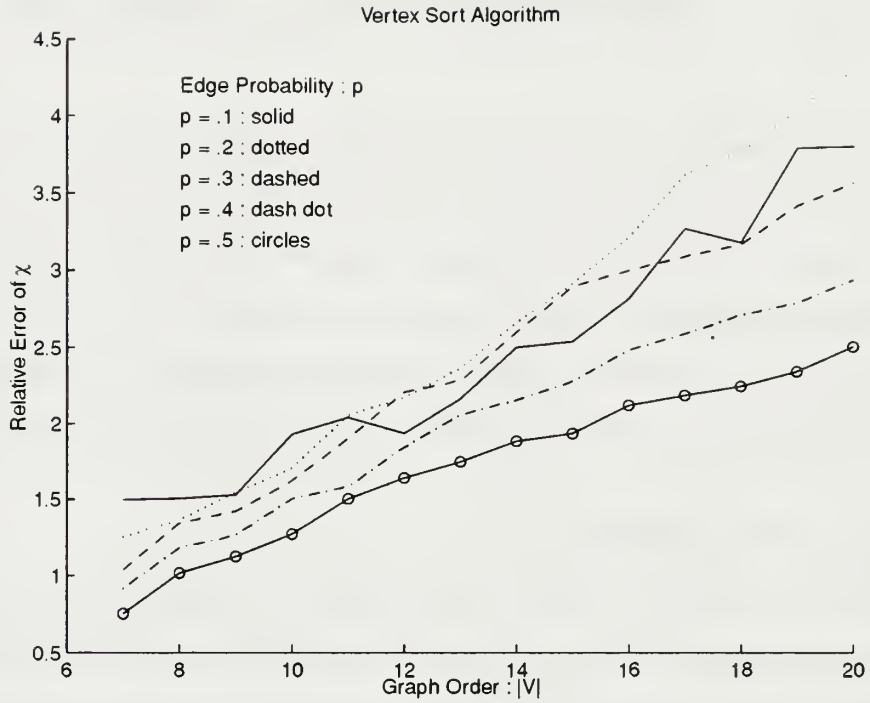

Figure 6. Relative error of χ versus graph order. Note: This is discrete data. Relative error has been represented by continuous lines for clarity only.

of large order tend to be harder to color due to the increased complexity of the graph. We are limited to graphs of order 20 due to the excessive use of computer time. The computation time of the function `gbcolor.m` to compute the exact $\chi(G)$ in graphs larger than order 20 was too excessive. The Missing Edge algorithm shows promise because the relative error is less than the other two algorithms even in the higher orders. On the other hand, the Vertex Sort algorithm does not perform as well in the lower orders as the other two algorithms, and does not show promise for further research. The Basic algorithm was used in this circumstance to provide a basis to the improved algorithms.

B. GRAPH DENSITY

The density of the graph is derived from the probability p that a given edge occurs. As can be seen from Table II and Figures 5 - 7, the density of the graph

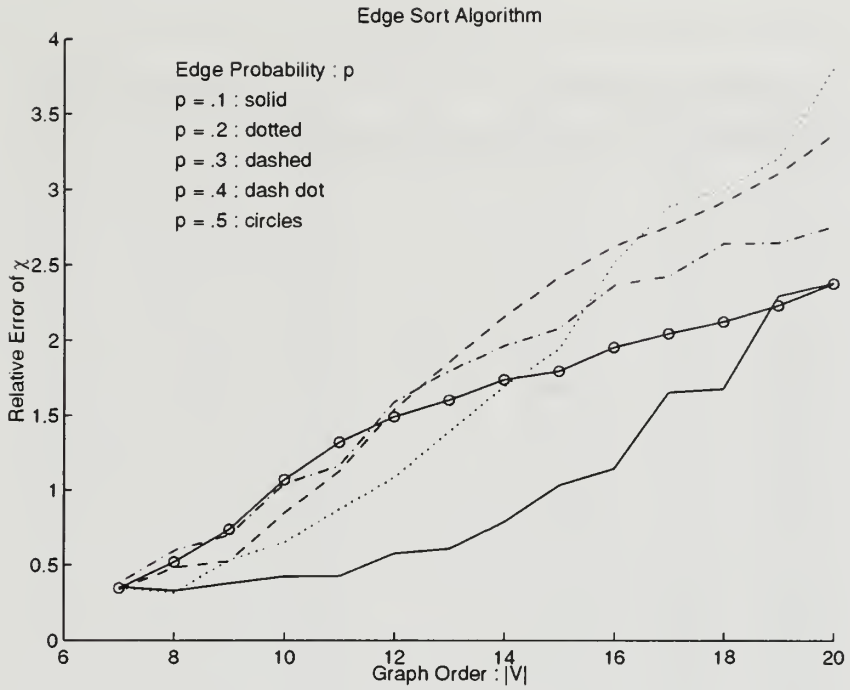

Figure 7. Relative error of χ versus graph order. Note: This is discrete data. Relative error has been represented by continuous lines for clarity only.

has a peculiar effect on the performance of all the algorithms. The edge probability $p = .2$ produces the largest error in all three algorithms at the higher order, and this error appears to be increasing faster than when other probabilities are used. There appears to be a peculiarity in the relative error of χ for graphs with edge probability between .1 and .3. Further research on the structure of random graphs with these edge probabilities is required to shed some light on this peculiarity. Again, the Vertex Sort algorithm shows little promise since the characteristics of Figure 6 shows no distinct pattern of the relative error in χ for different probabilities p. The Missing Edge algorithm shows promise especially in the probabilities $p = .1$ and $p = .5$.

C. CONCLUSION

In conclusion, we see that we have moved the NP-complete problem from the coloring of an arbitrary graph to the choice of the correct maximal chordal subgraph

with the perfect elimination order which will produce the minimum chordal supergraph of the arbitrary graph. If it were possible to choose this maximal chordal subgraph accurately the relative error of γ would be.

		Relative Error in χ					Relative Error in χ		
$\|V\|$	p	Proj 1	Proj 2	Proj 3	$\|V\|$	p	Proj 1	Proj 2	Proj 3
7	.1	0.7567	1.4967	0.3550	14	.1	2.1717	2.4967	0.7867
	.2	0.7250	1.2533	0.3483		.2	2.2308	2.6592	1.6900
	.3	0.6417	1.0400	0.3367		.3	2.3817	2.5975	2.1550
	.4	0.6775	0.9167	0.3800		.4	2.0807	2.1505	1.9615
	.5	0.5567	0.7528	0.3458		.5	1.8233	1.8813	1.7347
8	.1	0.8233	1.5050	0.3300	15	.1	2.0600	2.5350	1.0317
	.2	0.8525	1.3633	0.3142		.2	2.4900	2.9117	1.9367
	.3	0.8733	1.3425	0.4817		.3	2.5752	2.8927	2.4152
	.4	0.9392	1.1808	0.5958		.4	2.1815	2.2727	2.0757
	.5	0.7720	1.0157	0.5190		.5	1.8322	1.9313	1.7913
9	.1	1.1317	2.5350	0.3783	16	.1	2.4067	2.8142	1.1417
	.2	0.9533	2.9117	0.5317		.2	2.9792	3.2125	2.5150
	.3	1.0950	2.8927	0.5250		.3	2.8463	2.9962	2.6250
	.4	1.0377	2.2727	0.6973		.4	2.4402	2.4803	2.3655
	.5	0.9182	1.9313	0.7360		.5	2.0250	2.1163	1.9518
10	.1	1.1633	1.9267	0.4233	17	.1	2.3400	3.2717	1.6533
	.2	1.1608	1.7025	0.6500		.2	3.2992	3.6250	2.8958
	.3	1.2358	1.6183	0.8433		.3	2.9123	3.0895	2.7620
	.4	1.3308	1.5033	1.0408		.4	2.4982	2.5863	2.4300
	.5	1.2335	1.2707	1.0678		.5	2.1265	2.1827	2.0468
11	.1	1.3250	2.0375	0.4250	18	.1	2.3292	3.1825	1.6767
	.2	1.4575	2.0517	0.8692		.2	3.6058	3.7708	3.0058
	.3	1.5475	1.9000	1.1250		.3	3.0758	3.1702	2.9245
	.4	1.4802	1.5835	1.1617		.4	2.6853	2.7108	2.6468
	.5	1.4372	1.5022	1.3178		.5	2.1828	2.2433	2.1255
12	.1	1.5400	1.9325	0.5767	19	.1	3.2308	3.7942	2.2975
	.2	1.7483	2.1683	1.0833		.2	3.6065	4.0438	3.2207
	.3	1.9792	2.2050	1.5350		.3	3.2637	3.4208	3.1165
	.4	1.7282	1.8432	1.5872		.4	2.7017	2.7897	2.6507
	.5	1.5663	1.6390	1.4885		.5	2.1265	2.3386	2.2340
13	.1	1.6842	2.1600	0.6083	20	.1	3.1625	3.8025	2.3800
	.2	2.0250	2.3608	1.3875		.2	4.1508	4.2950	3.8142
	.3	2.0990	2.2798	1.8528		.3	3.4603	3.5655	3.3695
	.4	1.9398	2.0550	1.7923		.4	2.8642	2.9357	2.7575
	.5	1.6952	1.7450	1.5998		.5	2.1828	2.5000	2.3759

Table II. Experimental Results of the Supergraph Heuristic.

V. FURTHER RESEARCH

We have given a new approach to coloring an arbitrary graph with the use of a supergraph heuristic. With higher order graphs the relative error of the estimate is larger than anticipated, however the procedure for computing the estimate is a polynomial-time algorithm. The Edge Sort preprocessing algorithm shows promise of improving the relative error of the estimate and should be further studied for continued improvement. The main area for further research will be in the choice of the maximal chordal subgraph which will produce the minimum chordal supergraph. If this choice can be performed accurately our relative error would be next to nothing.

It follows from the work of Grone, Johnson, et al. that if G' is any chordal supergraph of G, and if H is the chordal subgraph of G produced by Maxchord or some related algorithm, then there exist a chordal completion sequence containing both H and G'. Finding such a sequence would be ideal. Since the algorithm for generating such sequences is completely driven by perfect elimination orderings, our problem reduces to that of finding an optimal perfect elimination ordering for H.

Further study is required in the analysis of the relative error of the estimate for the data in Table II and for larger graphs. To acquire data from larger graphs we must compile our programs into a more efficient programing language, possibly UNIX C.

APPENDIX A. PROGRAM FOR BASIC ALGORITHM

```
% Loren Eggen              Project #1                revised       30 May 97

p=.1;                                  % probability of edge present
fid=fopen('Results2/project1.out','a');
fprintf(fid,'Project #1\n');
fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
fprintf(fid,' Matrix  Ave. Chromatic Numbers  Elapsed\n');
fprintf(fid,'  Size    Actual  New   Error      Time\n');
fclose(fid);
for i=7:20                             % order of arbitrary graph G
  M=[];                                % initialize storage matrix
  k=0;                                 % initialize k
  t=clock;                             % start clock
  while k < 100;                       % generate k graphs
    v=[];                              % initialize v
    while length(v) ~= i               % loop for undirected connected graph
      G=unigraph1(i,p);                % generate a random undirected graph G
      [v,e]=connect1(G,1);             % check if G is connected
    end;                               % end while
    [v1,e1]=mchord(G,1);               % find the maximal chordal subgraph H
    H=mkadjmat(e1,i);                  % make adjancency matrix for H
    F=G-H;                             % determine missing edges
    em=mkedges(F); clear F;            % make a list of missing edges
    if ~isempty(em);                   % if the original graph was not chordal
      H1=H(v1,v1); clear H;            % reorder vertices of H to mchord peo-order
      em1=trans(v1,em);                % translate missing edges to new order
      c1=mclique(H1);                  % maximum clique of H1, lower bound of G
      vc1=gbcolor(G,c1);               % color G using greedy-backtracking coloring
      ac=max(vc1); clear G;            % chromatic number for G
      v2=mcs(H1,1);                    % find peo ordering using max. card. search
      H2=H1(v2,v2); clear H1;          % reorder vertices H1 to mcs peo-order
      em2=trans(v2,em1);               % translate missing edges to new order
      S=complete(H2,em2);              % complete H2 until missing edges are added
      c2=mclique(S); clear H2;         % maximum clique of S
      v=fliplr(1:length(S));           % reverse the ordering of S
      vc2=grcolor(S(v,v),c2);          % color S using greedy coloring
      nc=max(vc2); clear S;            % chromatic number for S
      M=[M;i ac nc (nc-ac)/ac];        % record results
```

```
      k=k+1;                          % increment k
    end;                              % end if
  end;                                % end while
  [n,m]=size(M);                      % size of M
  aac=sum(M(:,2))/n;                  % average actual chormatic number
  anc=sum(M(:,3))/n;                  % average new chormatic number
  adc=sum(M(:,4))/n;                  % average relative error
  tim=etime(clock,t)/60;              % elapsed time
  fid=fopen('Results2/project1.1.out','a');
  fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
  fprintf(fid,' Matrix    Chromatic Numbers\n');
  fprintf(fid,'  Size    Actual  New   Error\n');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f\n',M');
  fprintf(fid,'Average:%5.0f %7.0f %7.4f\n',aac,anc,adc);
  fclose(fid);
  fid=fopen('Results2/project1.out','a');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f %8.1f\n',i,aac,anc,adc,tim);
  fclose(fid);
end;                                  % end for
```

APPENDIX B. PROGRAM FOR THE VERTEX SORT ALGORITHM

```
% Loren Eggen              Project #2                    revised 30 May 97

p=.1;                                % probability of edge present
fid=fopen('Results2/project2.out','a');
fprintf(fid,'Project #2\n');
fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
fprintf(fid,' Matrix  Ave. Chromatic Numbers  Elapsed\n');
fprintf(fid,'  Size    Actual  New   Error      Time\n');
fclose(fid);
for i=7:20                           % order of arbitrary graph G
  M=[];                              % initialize storage matrix
  k=0;                               % initialize k
  t=clock;                           % start clock
  while k < 100                      % number of graphs to generate
    v=[];                            % initialize v
    while length(v) ~= i             % loop for undirected connected graph
      G=unigraph1(i,p);              % generate a random undirected graph G
      [v,e]=connect1(G,1);           % check if G is connected
    end;                             % end while
    [G1,v]=versort(G);               % sort the vertices by highest degree
    [v1,e1]=mchord(G1,1);            % find the maximal chordal subgraph H
    H=mkadjmat(e1,i);                % make adjancency matrix for H
    F=G1-H; clear G1;                % determine missing edges
    em=mkedges(F); clear F;          % make a list of missing edges
    if ~isempty(em);                 % if the original graph was not chordal
      H1=H(v1,v1); clear H;          % reorder vertices of H to mchord peo-order
      em1=trans(v1,em);              % translate missing edges to new order
      c1=mclique(H1);                % maximum clique of H1, lower bound of G
      vc1=gbcolor(G,c1);             % color G using greedy-backtracking coloring
      ac=max(vc1); clear G;          % chromatic number for G
      v2=mcs(H1,1);                  % find peo ordering using max. card. search
      H2=H1(v2,v2); clear H1;        % reorder vertices H1 to mcs peo-order
      em2=trans(v2,em1);             % translate missing edges to new order
      S=complete(H2,em2);            % complete H2 until missing edges are added
      c2=mclique(S); clear H2;       % maximum clique of S
      v3=fliplr(1:length(S));        % reverse the ordering of S
      vc2=grcolor(S(v3,v3),c2);      % color S using greedy coloring
      nc=max(vc2); clear S;          % chromatic number for S
```

```
      M=[M;i ac nc (nc-ac)/ac];      % record results
      k=k+1;                         % increment k
    end;                             % end if
  end;                               % end while
  [n,m]=size(M);                     % average actual chormatic number
  aac=sum(M(:,2))/n;                 % average new chormatic number
  anc=sum(M(:,3))/n;                 % average relative error
  adc=sum(M(:,4))/n;                 % average relative error
  tim=etime(clock,t)/60;             % elapsed time
  fid=fopen('Results2/project2.1.out','a');
  fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
  fprintf(fid,' Matrix    Chromatic Numbers\n');
  fprintf(fid,'  Size    Actual  New   Error\n');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f\n',M');
  fprintf(fid,'Average:%5.0f %7.0f %7.4f\n',aac,anc,adc);
  fclose(fid);
  fid=fopen('Results2/project2.out','a');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f %8.1f\n',i,aac,anc,adc,tim);
  fclose(fid);
end;                                 % end for
```

APPENDIX C. PROGRAM FOR MISSING EDGE ALGORITHM

```
% Loren Eggen              Project #3                    revised 30 May 97

p=.1;                                % probability of edge present
fid=fopen('Results2/project3.out','a');
fprintf(fid,'Project #3\n');
fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
fprintf(fid,' Matrix  Ave. Chromatic Numbers  Elapsed\n');
fprintf(fid,'  Size    Actual  New   Error      Time\n');
fclose(fid);
for i=7:20                           % order of arbitrary graph G
  M=[];                              % initialize storage matrix
  k=0;                               % initialize k
  t=clock;                           % start clock
  while k < 100                      % number of graphs to generate
    v=[];                            % initialize v
    while length(v) ~= i             % loop for undirected connected graph
      G=unigraph1(i,p);              % generate a random undirected graph G
      [v,e]=connect1(G,1);           % check if G is connected
    end;                             % end while
    [v1,e1]=mchord(G,1);             % find the maximal chordal subgraph H
    H=mkadjmat(e1,i);                % make adjancency matrix for H
    F=G-H;                           % determine missing edges
    em=mkedges(F); clear F;          % make a list of missing edges
    if ~isempty(em);                 % if the original graph was not chordal
      H1=H(v1,v1); clear H;          % reorder vertices of H to mchord peo-order
      em1=trans(v1,em);              % translate missing edges to new order
      c1=mclique(H1);                % maximum clique of H1, lower bound of G
      vc1=gbcolor(G,c1);             % color G using greedy-backtracking coloring
      ac=max(vc1);                   % chromatic number for G
      G1=G(v1,v1); clear G;          % translate G into H1 ordering
      F1=G1-H1; clear G1;            % find the max. edges missing
      [y,vs]=sort(sum(F1));          % sort by max. edges missing
      v2=fliplr(vs); clear F1;       % descending order
      H2=H1(v2,v2); clear H1;        % sort H1 max. edges missing first
      em2=trans(v2,em1);             % translate missing edges to new order
      v3=mcs(H2,1);                  % find peo ordering using max. card. search
      H3=H2(v3,v3); clear H2;        % reorder vertices H1 to mcs peo-order
      em3=trans(v3,em2);             % translate missing edges to new order
```

```
      S=complete(H3,em3);           % complete H2 until missing edges are adde
      c2=mclique(S); clear H3;      % maximum clique of S
      v4=fliplr(1:length(S));       % reverse the ordering of S
      vc2=grcolor(S(v4,v4),c2);     % color S using greedy coloring
      nc=max(vc2); clear S;         % chromatic number for S
      M=[M;i ac nc (nc-ac)/ac];     % record results
      k=k+1;                        % increment k
    end;                            % end if
  end;                              % end while
  [n,m]=size(M);                    % size of M
  aac=sum(M(:,2))/n;                % average actual chormatic number
  anc=sum(M(:,3))/n;                % average new chormatic number
  adc=sum(M(:,4))/n;                % average relative error
  tim=etime(clock,t)/60;            % elapsed time
  fid=fopen('Results2/project3.1.out','a');
  fprintf(fid,'Edge probability in Arbitrary Graph is:%4.1f\n',p);
  fprintf(fid,'Matrix     Chromatic Numbers\n');
  fprintf(fid,' Size    Actual   New   Error\n');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f\n',M');
  fprintf(fid,'Average:%5.0f %7.0f %7.4f\n',aac,anc,adc);
  fclose(fid);
  fid=fopen('Results2/project3.out','a');
  fprintf(fid,'%5.0f %7.0f %7.0f %7.4f %8.1f\n',i,aac,anc,adc,tim);
  fclose(fid);
end;                                % end for
```

APPENDIX D. GRAPH COMPLETION FUNCTION

```
function A=complete(A,e1)

%  function A=complete(A,e1)
%
%  This function is a graph completion function for the thesis project.
%  Input a peo ordering v, a list of edges e from a maximal chordal subgraph,
%  and a list of edges e1 necessary to make a super-hypergraph of the original
%  graph. The output is a chordal supergraph.

%  by Loren G. Eggen, 14 April, 1997.

A=A+speye(size(A));                          % add loops
[n,m]=size(A);                               % # of vertices
v=1:n;                                       % list of vertices
p=sum(A); q=find(p~=n);                      % eliminate and full
v=(v(q));                                    % vertices from list v
while ~isempty(e1)                           % loop for all missing edges
  k=length(v);                               % vertix by peo ordering
  l=max(find(~A(v(k),:)));                   % next highest missing edge
  if ~isempty(l)                             % test if found next
    ez=[v(k) l];                             % edge to add ez
    e=[e; ez];                               % add edge to list
    A(v(k),l)=1;                             % add edge in adjacency
    A(l,v(k))=1;                             % matrix
  end;                                       % end of if
  if sum(A(v(k),:))==n                       % test if vertix full
    h=find(v~=v(k));                         % eliminate vertix from
    v=v(h);                                  % list of vertices
  end;                                       % end of if
  [a,b]=size(e1);                            % a = length of missing edges
  for i=1:a                                  % for each missing edge
    if all(e1(i,:)==ez | e1(i,:)==fliplr(ez))  % test ez in missing edges
      if a==1                                % if yes and last then
        e1=[];                               % empty list of missing edges
        break;                               % break for loop
      else                                   % if not last
        x=1:a;                               % set x
        x=x(find(x~=i));                     % find missing edge = ez
```

```
        e1=e1(x',:);                    % delete edge from list
        break;                          % break for loop
      end;                              % end inner if
    end;                                % end outer if
  end;                                  % end for loop
end;                                    % end of while
A=A-speye(size(A));                     % eliminate loops
```

APPENDIX E. CONNECTED GRAPH FUNCTION

```
function [v,e]=connect1(A,i);

% function [v,e]=connect1(A,i);
%
%  This function will find a connected component in the input graph A and
%  starting vertex i. It uses depth first search and outputs the vertices
%  and edges of the connected component.
%

%  By Loren G. Eggen, 30 May, 1997.

v=i;                                    % initialize list of vertices
v0=i;                                   % first vertex
e=[];                                   % initialize list of deleted edges
n=length(A);                            % number of vertices
while length(v) < n                     % loop till all vertices are added
  x=adj(A,i);                           % adj. vertices to current vertex
  if ~isempty(x)                        % if x is not empty
    t=0;                                % test variable if vertex is added
    for j=1:length(x)                   % for all the adj. vertices
      if isempty(find(x(j)==v))         % find 1st one not in list v
        v=[v,x(j)];                     % add it to the list v
        e=[e; i,x(j)];                  % update deleted edges
        A(i,x(j))=0;                    % eliminate edge in adj. matrix
        A(x(j),i)=0;                    % eliminate edge in adj. matrix
        i=x(j);                         % make new vertex current
        t=1;                            % set test variable true
        break;                          % break when new one found
      end;                              % end of if
    end;                                % end of for
    if ~t                               % if no new vertex but x not empty
      for j=1:length(x)                 % for each adj. vertex which is on v
        e=[e;i,x(j)];                   % update deleted edges
        A(i,x(j))=0;                    % eliminate edge in adj. matrix
        A(x(j),i)=0;                    % eliminate edge in adj. matrix
      end;                              % end of for
      l=1;                              % set backtracking index
      while sum(A(i,:)) == 0 & i ~= v0 % backtrack till edge is present
```

```
        i=v(length(v)-1);                % backtrack list v
        l=l+1;                           % increment index
      end;                               % end of while
    end;                                 % end of if
  elseif i == v0                         % if x was empty and we returned to v0
    break;                               % break while loop, output component
  else                                   % not at the start but x is empty
    l=1;                                 % set backtracking index
    while sum(A(i,:)) == 0 & i ~= v0     % backtrack till edge is present
      i=v(length(v)-1);                  % backtrack list v
      l=l+1;                             % increment index
    end;                                 % end of while
  end;                                   % end of if
end;                                     % end of while
```

APPENDIX F. GREEDY-BACKTRACKING COLORING FUNCTION

```
function [vc]=gbcolor(A,m)

%  function [vc]=gbcolor(A,m)
%
%  This function uses the greedy-backtracking approach to
%  color the vertices of a graph so that no two colors are
%  together. Input the adjacency matrix of the graph and a
%  minimum number of colors. Output vc is the vector of vertix
%  colors.

%  by Loren G. Eggen, 18 March, 1997.

i=1;                                % starting index
v=i;                                % first vertex
vc=i;                               % first vertex color
n=length(A);                        % number of vertices
while length(vc) < n;               % used until all vertices have been colored
  k=1;                              % first color
  i=i+1;                            % increment index
  v=[v i];                          % vector of vertices visited
  x=adj(A(v,v),i);                  % adjacent visited vertices of index
  xc=sort(vc(x));                   % sorted colors
  for j=1:length(xc)                % find the next available color
    if xc(j) == k                   % if current color used
      k=k+1;                        % increment color
    end;                            % end if
  end;                              % end for
  if k > m                          % if we run out of colors backtrack
    i=i-1;                          % decrement index
    v=v(1:i);                       % go back one vertix
    vc=vc(1:i);                     % eliminate last color if necessary
    while length(v) > 1             % do not backtrack past 1
      t=0;                          % test variable to break backtrack
      while vc(i) < m               % if the color is < max see if we can
        vc(i)=vc(i)+1;              % increase the color
        if ~any(vc(i) ==  vc(adj(A(v,v),i)))    % test if the color has been
          t=1; break;                           % used if not use it and stop
        end;                        % end if
```

```
      end;                              % end while k < m
      if t == 1                         % found one that could be incremented
        break;                          % break outer loop
      end;                              % end if
      i=i-1;                            % decrement index
      v=v(1:i);                         % go back one vertix
      vc=vc(1:i);                       % eliminate last color if necessary
    end;                                % end while v > 1
  end;                                  % end if k > m
  if length(vc) < length(v)             % if the inner loop did not
    vc=[vc k];                          % update the color
  end;                                  % end if
  if length(v) == 1 & t == 0            % if we have ran out of colors and
    m=m+1;                              % backtracked to the origin increase
  end;                                  % the available colors
end;                                    % end while vc < n
```

APPENDIX G. GREEDY COLORING FUNCTION

```
function [vc]=grcolor(A,m)

%  function [vc]=grcolor(A,m)
%
%  This function uses the greedy approach to color the vertices of a graph
%  so that no two colors are together. Input the adjacency matrix of the graph
%  and a maximum number of colors. Output vc is the vector of vertix colors.
%  Optimal coloring if the input graph is chordal and reverse order perfect
%  elimination scheme.

%  by Loren G. Eggen, 23 April, 1997.

n=length(A);
k=1;
i=1;
v=i;
vc=k;
while length(v) < n;                % used until all vertices have been visited
  k=1;                              % first color
  i=i+1;                            % increment index
  v=[v i];                          % vector of vertices visited
  x=adj(A(v,v),i);                  % adjacent visited vertices of index
  xc=sort(vc(x));                   % sorted colors
  for j=1:length(xc)                % find the next available color
    if xc(j) == k                   % if color used
      k=k+1;                        % incremet color
    end;                            % end if
  end;                              % end for
  vc=[vc k];                        % update the color
  if max(vc) > m                    % if we have ran out of colors
    fprintf('colors used greater than colors given\n\n');
  end;                              % end if
end;                                % end while
```

APPENDIX H. MAXIMAL CHORDAL SUBGRAPH FUNCTION

```
function [v, E] = mchord(A,i)

%  function [v, E] = mchord(A,i)
%
%  Returns the peo ordering of vertices and a set of edges
%  which will generate a maximal chordal subgraph. Adjacency
%  matrix A should represent a connected undirected graph.
%  This function uses algorithm MAXCHORD, P.M. Dearing.

%  by Loren G. Eggen, 6 February, 1997

%  adj.m function called

%  begin mchord


v=i;                              % starting vertex v
n=length(A);                      % number of vertices
s=1:n;                            % list of vertices
s=s(find(s~=i));                  % delete first vertex from the list
C=zeros(n);                       % initialize the set of adj. vertices
C=sparse(C);                      % make matrix sparse
E=[];                             % initialize set of edges
for j=1:n-1                       % loop through all vertices except 1st
  [v1 e1]=adj(A,i);               % find adj. vertices to current
  for k=1:length(v1)              % loop through each adj. vertex u
    test = C(v1(k),:) | C(i,:);   % is set C(u) subset of C(v)
    if test == C(i,:)             % if so then
      C(v1(k),i)=1;               % update C(u)
      E=[E;i v1(k)];              % update set of edges
      A(i,v1(k))=0;               % delete edge from adjacency matrix
      A(v1(k),i)=0;               % both edges
    end;                          % end of if
  end;                            % end of inner for
  [l m]=max(sum(C(s,:)'));        % find next vertex with largest card. in C
  v=[v s(m)];                     % assign new vertex to reverse peo ordering
  i=s(m);                         % assign new vertex to current vertex v
  s=s(find(s~=i));                % delete v from list of vertices
```

```
end;                                    % end of outer for
v=fliplr(v);                            % peo ordering


% end of mchord
```

APPENDIX I. MAXIMUM CLIQUE FUNCTION

```
function c=mclique(H)

%  function c=mclique(H)
%
%   This function calculates the maximum clique number of a triangulated
%   graph H, which is ordered by it perfect elimination scheme.
%

%   by Loren G. Eggen,  21 April, 1997

%   calls adj.m function

c=1;                                % initial clique number
[n,m]=size(H);                      % order of input graph
S=zeros(1,n);                       % list, # if times vertices 1:n visited
a=1:n;                              % vertices 1:n
for i=1:n                           % loop for each vertex
  X=[];                             % initialize X
  v=a(i);                           % assign v current vertec
  [x,e]=adj(H,v);                   % find adj. vertices to v
  for j=1:length(x)                 % loop for each adj. vertex
    if find(v==a) < find(x(j)==a); % if index of v < index of adj. vertices add
      X=[X x(j)];                   % adj. vertex to the list of higher indices
    end;                            % end if
  end;                              % end innner for
  if isempty(x), v; end;            % if no adj. vertices v is cluque
  if ~isempty(X)                    % if X not empty
    u=min(X);                       % u, smallest index in X
    S(u)=max(S(u),length(X)-1);     % assign S(u) max. of current value or clique
    if S(v) < length(X)             % if S(v) < current clique
      [v X];                        % print current clique
      c=max(c,1+length(X));         % update maximum clique number
    end;                            % end inner if
  end;                              % end outer if
end;                                % end for loop
```

APPENDIX J. MAXIMUM CARDINALITY SEARCH

```
function [v] = mcs(A,i)

% function [v] = mcs(A,i)
%
%  Returns a vector of vertices which indicate
%  a possible perfect elimination scheme. Adjacency
%  matrix A should represent a connected undirected
%  graph. This function uses Maximum Cardinality Search.

%  by Loren G. Eggen, revised 29 January, 1997

%  adj.m function called

%  begin mcs


v=i;                    % 1st vertex in the peo
n=length(A);            % number of vertices
x=ones(1,n);            % initialize cardinality vector x
for j=1:n-1             % loop for each vertex
  v1=adj(A,i);          % find adj. vertices v1 to current vertex
  x(v1)=x(v1)+1;        % update the cardinality of vertices in v1
  [k,l]=max(x);         % find the vertex in x with max. cardinality
  v=[v l];              % add new vertex to the peo list v
  x(v)=x(v)-x(v);       % zero the entries of x for vertices in v
  A(i,l)=0;             % eliminate edges from adj. matrix
  A(l,i)=0;             % eliminate edges from adj. matrix
  i=l;                  % assign current vertex to the last vertex added
end;                    % end for loop
v=fliplr(v);            % reverse peo ordering


% end of mcs
```

APPENDIX K. MISCELLANEOUS FUNCTIONS

```
function [v, e] = adj(A,i)

%  function [v, e] = adj(A,i)
%
%  Returns a vector of adjacent vertices and a list of
%  edges to the vertex i from the adjacency matrix A.
%

%  by Loren G. Eggen, 29 January, 1997

%  no intrinsic functions called

%  begin adj

v=[];                   % initialize v
e=[];                   % initialize e
for j=1:length(A)       % for each element in row i
  if A(i,j)             % if an edge exist
    v=[v j];            % update v, list of adjancect vertices
    e=[e;i j];          % update e, list of associated edges
  end;                  % end if
end;                    % end for

% end of adj

function A=mkadjmat(e,v)

%  function A=mkadjmat(e,v)
%
%  Input a set of edges containing the numeric vertices e.g.
%  e=[1 2;1 7;2 3;2 5;3 4;4 6;5 6;6 7] and maximum vertices v.
%  Output an adjacency matrix of unidirectional graph.
%

%  by Loren G. Eggen, revised 3 May, 1997

A=zeros(v,v);
[n,m]=size(e);
```

```
for i=1:n
  A(e(i,1),e(i,2))=1;
  A(e(i,2),e(i,1))=1;
end;
A=sparse(A);

function e=mkedges(A)

%  function e=mkedges(A)
%
%  This function makes a set of edges from the given
%  adjancency matrix.

%  by Loren G. Eggen, 11 April, 97

A=triu(A);
for i=1:length(A)
  y=find(A(i,:));
  e=[e;i*ones(length(y),1),y];
end;

function e1=trans(v,e)

%  function e1=trans(v,e)
%
%  This function translates the edges into the new vertix ordering.

%  by Loren G. Eggen, 14 April, 1997.

[n,m]=size(e);
for i=1:n
  e1(i,1)=find(e(i,1)==v);                    % translate the missing edges
  e1(i,2)=find(e(i,2)==v);                    % into the new ordering
end;

function [A] = unigrap1(n,p)

%  function[A] = unigrap1(n,p)
%
%  Generates an edge with probability p in an adjacency
%  matrix for a unidirected graph.
```

```
%  by Loren G. Eggen, revised 29 May, 1997

%  no intrinsic functions called

% begin

A=rand(n);                              % generate random 0-1 matrix nxn
A=A(:,:)<p;                             % eliminate all entries > p
A=A-diag(diag(A));                      % eliminate diaganol
A=triu(A);                              % eliminate lower triangular
A=A+A';                                 % make symetric
A=sparse(A);                            % make matrix sparse storage

% end

function [S,v]=versort(A);

% function [S,v]=versort(A);
%
%  Label an adjacency matrix sorting by highest degree
%  vertix in the matrix.

i=sum(A);
[y,j]=sort(i);
v=fliplr(j);
S=A(v,v);
```

LIST OF REFERENCES

[1] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* American Elsevier, New York, 1976.

[2] D. B. West. *Introduction to Graph Theory.* Prentice-Hall Inc., Upper Saddle River, NJ 07458, 1996.

[3] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[4] Xue J. Edge-maximal triangulated subgraphs and heuristics for the maximum clique problem. *NETWORKS*, 24:109–120, 1994.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithm.* McGraw-Hill, New York, 1990.

[6] M.R Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[7] O. Ore. *The Four-Color Problem.* Academic Press, New York, 1967.

[8] E. Balas and C.S. Yu. Finding the maximum clique in an arbitrary graph. *SIAM J. Computing*, 15(4):1054–1068, 1986.

[9] E. Balas and J. Xue. Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM J. Computing*, 20(2):209–221, 1991.

[10] P.M. Dearing, D.R. Shier, and D.D. Warner. Maximal chordal subgraphs. *Discrete Applied Mathematics*, 20:181–190, 1988.

[11] F. S. Roberts. *Applied Combinatorics.* Prentice Hall, Englewood Cliffs, NJ., 1984.

[12] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[13] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum, covering by cliques, and maximum indepentent set of a chordal graph. *SIAM J. Computing*, 1(2):180–187, 1972.

[14] E. A. Bender and Wilf H. S. A theoretical analysis of backtracking in the graph coloring problem. *Journal of Algorithms*, 6:275–282, 1985.

[15] R. E. Tarjan and Yannakakis M. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectivily reduce acyclic hypergraphs. *SIAM J. Computing*, 13(3):566–579, 1984.

[16] Grone R., Johnson C. R., and Wolkowicz H. Positive definite completions of partial hermitian matrices. *Linear Algebra and its Applications*, 58:109–124, 1984.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dryer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code MA 1
   Department of Mathematics
   Naval Postgraduate School
   1411 Cunningham Road, Rm 341
   Monterey, Ca 93943-5216

4. Professor Craig W. Rasmussen, Code MA/Ra 3
   Department of Mathematics
   Naval Postgraduate School
   1411 Cunningham Road, Rm 341
   Monterey, Ca 93943-5216

5. Captain Loren G. Eggen 2
   United States Military Academy
   Department of Mathematical Sciences
   P.O. Box 229
   West Point, NY 10996